METHOD AND APPARATUS FOR ATOMIC FILE LOOK-UP

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application claims priority from co-pending provisional patent application serial number 60/227,510, filed August 24, 2000 by the inventor hereof, the entire disclosure of which is incorporated herein by reference.

[0002] Much of what is disclosed in this application is also disclosed in application serial numbers, __/___,___, entitled, "Method and Apparatus for Handling Communication Requests at a Server Without Context Switching," and __/___,___, entitled, "Embedded Protocol Objects," both filed on even date herewith and assigned to the assignee of the present application, both of which are incorporated herein by reference.

CROSS-REFERENCE TO COMPUTER PROGRAM LISTING APPENDIX

[0003] A portion of the present disclosure is contained in a compact disc, computer program listing appendix. The compact disc contains an MS-DOS file entitled "tux2-full.txt" created on June 15, 2001, of 1,057 kilobytes. The contents of this file are incorporated herein by reference. Any references to "the appendix" or the like in this specification refer to the file contained on the compact disc.

[0004] The contents of this file are subject to copyright protection. The copyright owner has no objection to the reproduction by anyone of the patent document or the appendix as it appears in the Patent and Trademark Office patent files or records, but does not waive any other copyright rights by virtue of this patent application.

BACKGROUND

[0005] The dramatic increase in usage of personal computers, workstations, and servers to perform every sort of daily task in recent decades has meant that the speed at which computer systems operate has become very important. The speed at which a computer system carries out the execution of an application is largely dependent on the speed of the operating system. An operating system in a modern computer system controls and monitors the use of hardware resources and provides programming interfaces to applications.

[0006] One process that slows down the performance of an operating system and the applications installed on a computer system is the process of suspending the execution of the operating system to conduct certain input and output (I/O) related operations. Such a suspension often occurs when files are opened. In order to open a file, an operating system must determine a "path" to the file by accessing a namespace in the file system. Accessing the namespace in the file system suspends execution of operating system tasks. However, once a file has been accessed the first time, the path for the file is typically cached in memory, so that future accesses of the file, at least for a time, do not cause the operating system to suspend execution of tasks. An operation that does not cause the operating system to suspend the execution of tasks is often referred to as an "atomic" operation.

[0007] Applications do not have a way, at execution time, of knowing if a file system path is stored in memory or not. An application must simply ask the operating system to open the file. An application developer must simply live with the possi-

bility that a serious performance or scheduling impact will result if the operating system must determine the file path from the file system namespace, and hope that at least in some cases, the opening of the file will be able to proceed atomically.

SUMMARY

[0008] The present invention provides for an application to find out whether a file is being opened atomically based on whether or not the file path is present in a file system namespace cache. By making use of this feature, an application developer can design an application to determine ahead of time if the file operation will cause the task execution to be interrupted, and react accordingly. In many cases, if the file cannot be opened atomically, the application can be designed to redirect the file operation request to another process, which may include blocking point handling. Through the use of this feature, impacts to the scheduling of various operations involved in executing an application can be minimized or avoided.

[0009] According to one embodiment of the invention, an operating system includes a kernel, a user space, and a file system. The operating system responds to file operation requests relative to a specific file received from an application by determining if a file path corresponding to the specific file is stored in a file system namespace cache. The operating system notifies the application that the file operation was not, could not, or is not being performed atomically if the file path is not stored in the file system namespace cache. If the file cannot be opened atomically, the application can redirect the request. The request can be redirected by the application to a process that includes blocking point handling. The blocking point handling

can be either in the kernel or in the user space. If the file path is cached in the file system namespace cache, the file operation is performed, for example, the file is simply opened, and the application is notified that the file has been opened atomically so that the file can be used without a scheduling impact.

[0010] In one embodiment, an operating system according to the present invention includes a file system including a file system namespace, and a user space operatively connected to the file system namespace. The user space is operative to enable the execution of at least one application. An operating system kernel is operatively connected to the user space and the file system.

[0011] The operating system kernel includes the file system namespace cache for caching file paths from the file system namespace. The operating system kernel also includes what is referred to herein as an atomic look-up operation. The atomic look-up operation is the function that is called when an application seeks to open a file, or perform a similar file operation. The atomic look-up operation determines if the specific file path corresponding to a file is stored in the file system namespace cache. If the file path is not stored in the cache, the atomic look-up operation notifies the application that the specific file was not opened atomically and the application can handle the situation appropriately.

[0012] In example embodiments of the invention, computer program code is used to implement many aspects of the invention. The computer program can be stored on a medium. The medium can be magnetic, such as a diskette, tape, or fixed disk, or optical, such as a CD-ROM or DVD-ROM. The computer program code can also be stored in a semiconductor device. Additionally, the computer program can be

supplied via the Internet or some other type of network. A workstation or computer system that is connected to a network typically runs the computer program code supplied as part of a computer program product. This computer system can also be called a "program execution system" or "instruction execution system." The computer program code in combination with the computer system and measurement system forms the means to execute the method of the invention.

BRIEF DESCRIPTION OF THE DRAWINGS

[0013] FIG. 1 is a block diagram that illustrates some features of software that implements an embodiment of the invention.

[0014] FIG. 2 is a flowchart illustrating a method performed by software that implements one embodiment of the invention.

[0015] FIG. 3 is another flowchart illustrating a method performed by software that implements an embodiment of the invention.

[0016] FIG. 4 is a block diagram that illustrates some features of software that implements an embodiment of the invention.

[0017] FIG. 5 is a flowchart illustrating a method performed by software that implements one embodiment of the invention.

[0018] FIG. 6 is a block diagram of a computer system that is implementing an embodiment of the invention.

DETAILED DESCRIPTION OF ONE OR MORE EMBODIMENTS

[0019] The present invention is typically embodied in computer software or a computer program product. It should be understood that not every feature of the software described is necessary to implement the invention as claimed in any particular one of the appended claims. The complete software product is described rather to fully enable the invention. It should also be understood that throughout this disclosure, where a software process or method is shown or described, the steps of the method may be performed in any order or simultaneously, unless it is clear from the context that one step depends on another being performed first.

[0020] The embodiments of the present invention described are implemented in a computing platform based on the computer operating system commonly known as "Linux" that is available as open source directly over the Internet. Linux is also available through various vendors who provide service and support for the Linux operating system. Among these vendors are Red Hat, Inc., of Research Triangle Park, North Carolina, the assignee of the present invention. An example of computer program code in patch format that implements the invention is included in the appendix, and its use will be discussed later. Certain, more brief code samples are included in this specification to illustrate specific concepts where discussed. All of these examples will be readily understood by those of ordinary skill in the art. It will also be understood that Linux examples are shown for illustrative purposes only. The inventive concepts described herein can be adapted to any computing platform based on any operating system, including those based on Macintosh™, Unix™ and Windows™.

[0021] Finally, it should be understood the several block diagrams and flow-charts which are used to illustrate the inventive concepts are not mutually exclusive. Rather, each one has been tailored to illustrate a specific concept discussed. In many cases, the elements or steps shown in a particular drawing co-exist with others shown in a different drawing, but only certain elements or steps are shown for clarity. For example, the block diagrams of Figures 1 and 4 both show elements within an operating system kernel and user space. In actual software, all of the elements in both the drawings might be present. However, only the ones relevant to a particular feature are shown in each drawing for clarity.

[0022] Turning to FIG. 1, a block diagram is presented that illustrates various elements of a software system having an operating system kernel, 100, and a user space, 101. The features that are illustrated by FIG. 1 are important to a computer platform or instruction execution system to be used as a server, and so the software system of FIG. 1 can be referred to as a "server system." The operating system kernel, or simply, "the kernel" is the part of the operating system software that handles hardware resources, provides fundamental functionality, and provides fundamental programming interfaces to applications. Such a programming interface is often referred to as an "application programming interface" or "API." In the present embodiment of the invention, the operating system kernel, 100, includes the capability to maintain a communication protocol stack through the use of an in-kernel, application protocol subsystem, 102. It is important to distinguish an application protocol such as HTTP or FTP from lower level protocols such as TCP/IP. Trusted protocol modules 103 are also included in the kernel and provide application protocol information and

functionality to the protocol subsystem, 102. The protocols involved can be HTTP, FTP, or any other application protocol used for network communications, including so-called "meta application protocols" such as extended markup language (XML) and hypertext markup language (HTML), which use HTTP. It is to be understood that references to HTTP included herein are meant to include HTML and XML. A generic operating system cache, 104, also resides in the kernel and can be used to cache files or pages of information. Finally, a protocol object cache, 105, is also in the kernel and is operatively connected to protocol subsystem 102. The protocol object cache can be important to certain features of the software system to be discussed later. Note that, as shown in FIG. 1, the protocol subsystem in this example embodiment provides a direct common gateway interface (CGI) and transparent socket redirection.

[0023] It cannot be overemphasized that the operating system architecture discussed with FIG. 1 (and that discussed later with reference to FIG. 4) are representative examples. Some operating systems allow certain applications to be run in kernel space, but others to be run in user space. It is also possible in some circumstances for an application to run in both places, with some code running in kernel space, and other code running in user space.

[0024] User space 101 of FIG. 1 contains untrusted modules or other executables, 106. It is beneficial at this point to explain what is meant by "trusted" vs. "untrusted" software modules. As is known by those skilled in the art, most modern operating systems are designed so that applications or the software modules residing in user space are in a "sandbox" of sorts. This sandbox guarantees that modules in

the operating system cannot be corrupted or adversely affected by what goes on in user space. This concept is implemented by the designing the operating system so that modules in user space are considered "untrusted" such that their access to operating system functions is limited. Modules inside the operating system kernel, by contrast, are "trusted" and have full access to operating system functions. In this example embodiment of the invention, high-level, communication protocol applications can reside as trusted modules inside the operating system kernel. These in-kernel protocol modules and the in-kernel protocol subsystem enable a server system to respond to application protocol requests without the operating system switching contexts. It is important to note that not all operating systems have a separate user space. Some operating systems execute applications within the operating system kernel space. In these cases however, the functional relationship between the operating system and the application is the same. The operating system handles hardware and provides API's, and the application uses an API to perform application tasks.

[0025] Operations in a computer system that do not cause context switching or any other processing schedule interruptions are often referred to as "atomic" operations. With the architecture just described, communications applications can do what would normally be "non-atomic" work without context switching. When a response needs to be generated, information about the application protocol request is stored in an in-memory, in-kernel request structure which enables the kernel to resume execution once a commensurate user space request structure is updated as if the response has already taken place. An example request structure for an HTTP

request is shown below. In this particular case, the user request structure is shown; however, the kernel request structure is very similar, and can be easily derived from the user space request structure.

```
typedef struct user_req_s {
        int version_major;
        int version_minor;
        int version_patch;

        int http_version;
        int http_method;

        int sock;
        int event;
        int thread_nr;
        void *id;
        void *priv;

        int http_status;
        int bytes_sent;
        char *object_addr;
        int module_index;
        char modulename[MAX_MODULENAME_LEN];

        unsigned int client_host;
        unsigned int objectlen;
        char query[MAX_URI_LEN];
        char objectname[MAX_URI_LEN];

        unsigned int cookies_len;
        char cookies[MAX_COOKIE_LEN];

        char content_type[MAX_FIELD_LEN];
        char user_agent[MAX_FIELD_LEN];
        char accept[MAX_FIELD_LEN];
        char accept_charset[MAX_FIELD_LEN];
        char accept_encoding[MAX_FIELD_LEN];
        char accept_language[MAX_FIELD_LEN];
        char cache_control[MAX_FIELD_LEN];
        char if_modified_since[MAX_FIELD_LEN];
        char negotiate[MAX_FIELD_LEN];
        char pragma[MAX_FIELD_LEN];
        char referer[MAX_FIELD_LEN];

        char *post_data;
        char new_date[DATE_LEN];
} user_req_t;
```

[0026] FIG. 2 is a flowchart illustrating the method of responding to a request without context switching. At step 201 the user space request structure is created. At step 202, execution by the instruction execution system operating as a server system enters the operating system kernel. At step 203, a new kernel request structure is created to handle the request. The kernel request structure is populated with data received in conjunction with the request, usually from a client system making the request over the network. At step 204, the kernel request structure is copied to the user space by populating the user space request structure. This "copy" is most efficiently done by moving data related only to used portions of the request structures. In this case, a so-called, "private field" of the request structure would also be moved. The private field is an opaque pointer that can be used by applications for reference. At step 205, execution returns to user space, and the user space application continues operations as if the request has been handled. The user space request structure will be overwritten if another request needs to be handled. The kernel handles the request normally in due course (this step is not shown for clarity). At 206, the operating system monitors the request. If the request is completed, or a time out occurs, execution enters the operating system kernel at step 207, and the kernel request structure is deleted at 208 to save memory resources. If the request cannot be completed, but a time out has not occurred, the system will maintain the request and the request will be returned as the current request to user space after the I/O operation(s) for that request has finished. A "suspended" request will not cause execution to be suspended. It should be noted that the request is suspended – not the task, so

that execution continues in user space. It is also possible for the request to be suspended on more than one I/O operation.

[0027] FIG. 3 illustrates another process included in the operating system that embodies the invention; the method of embedding static protocol objects in responses, such as HTTP Web pages, sent to a client application in response to an application protocol request. It should be noted that this illustrating is based on HTTP Web pages by way of example only. It may be possible to implement the invention with other application protocols, such as FTP. In this embodiment of the invention, the static protocol objects are stored in the protocol object cache, 105, of FIG. 1, but they can also reside in the generic operating system cache. At step 301 of FIG. 3 the server receives and analyses the application protocol request. A sample application protocol request in the form of an HTTP request is shown below. Such a request is normally terminated by two newline characters.

```
GET / HTTP/1.0
 User-Agent: Wget/1.6
 Host: www.redhat.com
 Accept: */*
```

At step 302 a preamble is sent back to the client. The preamble is the header portion of an HTTP response or reply. An example header portion is shown below.

```
HTTP/1.1 200 OK
 Date: Tue, 03 Jul 2001 10:45:31 GMT
 Server: Apache/1.3.19 (Unix)  (Red-Hat/Linux) mod_ssl/2.8.1
OpenSSL/0.9.5a
 Connection: close
 Content-Type: text/html
```

[0028] At step 303 the optional dynamic portions of the reply page are created within a memory buffer on the server. The dynamic portions are made up of dynamic protocol objects, which form part of the reply. The reply can also have static protocol objects embedded within. The dynamic protocol objects are sent to the client application at step 304. At step 305 the static parts of the reply page, or static protocol objects are retrieved from the protocol object cache or the generic operating system cache. The static objects are sent to the client application at step 306, where they are embedded in the reply so that a complete reply can be displayed at the client system

[0029] FIG. 4 is another block diagram of an embodiment of the operating system of the present invention. In this case, the block diagram is constructed to illustrate the operation of the atomic file look-up feature. FIG. 4 includes three main segments of an operating system environment: file system 400, user space 401, and operating system kernel 402. File system 400 includes file system objects 403 in the file system space. File system objects, or simply "files" are the building blocks of the file system space. In effect, they are containers of information. There are various types of file system objects, including regular files, directories, and symbolic links. A symbolic link provides a way to create a new file that in fact just points to an existing file. A symbolic link is also referred to as a soft link. The linking is done via a string including separator characters. The separator is the "/" (slash) character on Linux and Unix systems and for Internet URL's, and it is the "\" (backslash) character on Windows systems.

[0030] Note that the file system object name or file name is a string of characters which do not include the separator character, for example, "homework.doc". Every file system object has a file system object name. A file system object path or "file path" however is a string of characters which usually start with the separator character, and specify various parts of a path which will locate the file. In some systems this file path can also be called a directory path or folder path, for example, "/home/joe/docs/homework.doc".

[0031] The file system of FIG. 4 also includes the file system namespace, 404. The file system namespace refers to all file paths within the system. The paths for all the files are typically stored on the file system device in the form of directories. The user space, 401, shown in FIG. 4 is where applications 405 typically reside. These applications make use of the file system objects, 403, that are kept within file system 400. In order to make use of a not-yet-used file, the application and operating system perform a file system namespace operation. Such an operation is performed, for example, to open, read or write to a file, rename a file, or move it between directories. There are other, well-known file system namespace operations as well. For purposes of this disclosure, any of these file system namespace operations are generically referred to as "opening a file", processing a "file open request", performing a file operation, or the like. In the prior art, if an application wanted to read the first 1000 bytes of the "/home/joe/docs/homework.doc" file, it would contain computer program code similar to what is shown in the following C example.

```
fd = open("/home/joe/docs/homework.doc", O_RDONLY);
read(fd, buf, 1000);
close(fd);
```

After executing these three system-calls, the operating system kernel will have put the first 1000 bytes of the file into the memory buffer, "buf".

[0032] The operating system kernel includes a file system namespace cache, 406. The "open" system-call, as shown above, will access the file system name-space cache internally, to speed up access to files, once their path is stored in the file system namespace cache. The file system name space cache is also sometimes called a "dentry cache". Note that the namespace cache does not cache file contents. It might cache file attributes. File contents are cached in a separate data structure, such as the generic cache shown in FIG. 1, often called the "page cache", as is well-known and has been omitted from FIG. 4 for clarity.

[0033] In the prior art as illustrated above, the kernel will get to the file whatever it takes in terms of time and resources, and it is transparent to the application whether the file path with the file name was already present in the namespace cache or not. If not, the kernel would read file path details from disk, and the processing associated with the application will be suspended and context switches occur until that I/O has finished, creating a serious performance impact.

[0034] In contrast to the prior art just described, the system of FIG. 4 includes an atomic look-up operation, 407, which implements the atomic file look-up according to this embodiment of the invention. The atomic file lookup empowers an application, 405, to detect whether a file is already cached within the namespace cache, or not. The following code shows how the feature is used in practice.

```
fd = open("/home/joe/docs/homework.doc", O_RDONLY | O_ATOMICLOOKUP);
```

```
if (fd == -EWOULDBLOCKIO) {
   /*
    * The file name was not yet cached, bounce this open()to another
    * process:
    */
    return -1;
}
read(fd, buf, 1000);
close(fd);
```

Note the "O_ATOMICLOOKUP" flag and the new "-EWOULDBLOCKIO" return code for the open system-call. Upon learning that the file path is not yet cached, the application can bounce or redirect the file open to another process, as determined appropriate by the application developer. In some cases this process may be a process that performs blocking point handling.

[0035] The blocking point handling process can reside in the user space at 408 of FIG. 4, or in the kernel at 409 of FIG. 4. A "blocking point" is a point where a lookup would cause a possible context switch. "Handling the blocking point" might involve bouncing this lookup to another process/thread, which can manage the file open in a way that improves performance.

[0036] FIG. 5 illustrates one example of the atomic look-up operation in flowchart form. At step 501 an application determines what file name, as designated by a character string, is to be opened. At 502 the file open request is made to the operating system. At step 500 the operating system kernel receives the request and attempts to open the file atomically. In order to attempt the atomic open, the kernel checks the file system namespace cache for the appropriate file path. Step 503 is a decision point that checks for errors such as the lack of existence of such a file. This process is handled in the same way as in the prior art. If there is an error, it is han-

dled by an error handling routine at 504. If there is no error, processing continues. The operating system kernel notifies the application at step 505 as to whether the file was, or is being opened atomically because its path was stored in the file system namespace cache. If the file was opened atomically, the file is used at step 506. If not, since the application has been notified of the fact that an atomic open was not possible at step 505, a possible blocking point is handled at step 507.

[0037] Note that in this embodiment, the application does not send an inquiry as to whether the file path is cached. It simply notifies the operating system kernel that it wants to open a file atomically. If the kernel can comply because the file path is cached, the file is opened and a handle is returned to the application. If the file path is not cached, then the file is not opened, and –EWOULDBLOCKIO is returned to the application. With respect to the discussion herein of this feature, the terminology "was opened", "could be opened", "is being opened" and the like is used interchangeably. Likewise, the terminology regarding a file operation such as "was performed", "is being performed", "could be performed" and the like is used interchangeably. The operation of the invention is the same regardless of the exact timing of the checking of the cache, the file opening, and the notification from the operating system kernel to an application.

[0038] There are many variations of the operating system architecture described above, any of which, can also include the atomic look-up operation. For example, the file system device does not have to reside on the computer system that is maintaining the operating system kernel, but can instead be accessed over a network. In this case, the file system namespace and be located with the networked file

system device, located on the system that is maintaining the operating system kernel, or distributed in both places. It is also important to note that not all operating system environments have a user space as a separate protection domain. In some operating systems, for example, those commonly used with embedded processors, applications are maintained and executed within the operating system kernel. In this case, the operating system still provides API's for applications, and the operating system still communicates with an application through its API, just the same as if the application were running in a user space.

[0039] Included at the end of this specification before the claims is one example of a source code listing showing code that implements the atomic look-up feature. This source code listing is entitled, "Source Code Listing Submitted as Part of the Specification." The source code is in the well-known differential patch format. It patches the publicly available version 2.4.2 of the Linux operating system, an open source operating system that can be acquired over the Internet, and from companies that provide support for it, such as Red Hat, Inc., the assignee of the present application. The code contained in the source code appendix for the present application, discussed below, also implements the atomic look-up feature.

[0040] As previously discussed, in some embodiments, the invention is implemented through computer program code operating on a programmable computer system or instruction execution system such as a personal computer or workstation, or other microprocessor-based platform. FIG. 6 illustrates further detail of a computer system that is implementing the invention in this way. System bus 601 interconnects the major components. The system is controlled by microprocessor 602,

which serves as the central processing unit (CPU) for the system. System memory 605 is typically divided into multiple types of memory or memory areas such as read-only memory (ROM), random-access memory (RAM) and others. The system memory may also contain a basic input/output system (BIOS). A plurality of general input/output (I/O) adapters or devices, 606, are present. Only three are shown for clarity. These connect to various devices including a fixed disk drive, 607, a diskette drive, 608, network, 610, and a display, 609. Computer program code instructions for implementing the functions of the invention are stored on the fixed disk, 607. When the system is operating, the instructions are partially loaded into memory, 605 and executed by microprocessor 602. Optionally, one of the I/O devices is a network adapter or modem for connection to network, 610, which may be the Internet. It should be noted that the system of FIG. 6 is meant as an illustrative example only. Numerous types of general-purpose computer systems are available and can be used.

[0041] Elements of the invention may be embodied in hardware and/or software as a computer program code (including firmware, resident software, microcode, etc.). Furthermore, the invention may take the form of a computer program product on a computer-usable or computer-readable storage medium having computer-usable or computer-readable program code embodied in the medium for use by or in connection with an instruction execution system such as that shown in FIG. 6. Such a medium is illustrated graphically in FIG. 6 to represent the diskette drive. A computer-usable or computer-readable medium may be any medium that can contain, store, communicate, or transport the program for use by or in connection with an in-

struction execution system. The computer-usable or computer-readable medium, for example, an electronic, magnetic, optical, electromagnetic, infrared, or semiconductor system. The medium may also be simply a stream of information being retrieved when the computer program product is "downloaded" through a network such as the Internet. Note that the computer-usable or computer-readable medium could even be paper or another suitable medium upon which a program is printed.

[0042] The appendix to this application includes source code in differential patch format that implements the features described in this specification. The source code is intended to patch a version of the Linux operating system, an open source operating system that can be acquired over the Internet, and from companies that provide support for it, such as Red Hat, Inc., the assignee of the present application. The code is intended to patch version 2.4.5 of the Linux operating system, which has already been patched by the well-known, so-called, "ac4" patch. Version 2.4.5 of Linux is available, among other places, at: www.kernel.org/pub/linux/kernel/v2.4/linux-2.4.5.tar.gz. The above-mentioned ac4 patch is available, among other places, at: www.kernel.org/pub/linux/kernel/people/alan/2.4/patch-2.4.5-ac4.gz. A copy of a Linux operating system that includes all the features described herein can be constructed by downloading the files mentioned above, and entering, on a Linux system:

```
tar xzf linux-2.4.5.tar.gz
    gzip -d patch-2.4.5-ac4.gz
    cd linux
    patch -p1 < ../patch-2.4.5-ac4
    cd ..
    patch -p0 < tux2-full.txt
```

[0043] Specific embodiments of an invention are described herein. One of ordinary skill in the computing arts will quickly recognize that the invention has other applications in other environments. In fact, many embodiments and implementations are possible. The following claims are in no way intended to limit the scope of the invention to the specific embodiments described above.

SOURCE CODE LISTING SUBMITTED AS PART OF THE SPECIFICATION

```
--- linux/fs/namei.c.origTue Apr 17 13:18:46 2001
+++ linux/fs/namei.c Tue Apr 17 13:18:52 2001
@@ -423,9 +423,13 @@
 {
     struct dentry *dentry;
     struct inode *inode;
-    int err;
+    int err, atomic;
     unsigned int lookup_flags = nd->flags;

+    atomic = 0;
+    if (lookup_flags & LOOKUP_ATOMIC)
+        atomic = 1;
+
     while (*name=='/')
         name++;
     if (!*name)
@@ -494,6 +498,9 @@
         /* This does the actual lookups.. */
         dentry = cached_lookup(nd->dentry, &this, LOOKUP_CONTINUE);
         if (!dentry) {
+            err = -EWOULDBLOCKIO;
+            if (atomic)
+                break;
             dentry = real_lookup(nd->dentry, &this, LOOKUP_CONTINUE);
             err = PTR_ERR(dentry);
             if (IS_ERR(dentry))
@@ -557,6 +564,9 @@
         }
         dentry = cached_lookup(nd->dentry, &this, 0);
         if (!dentry) {
+            err = -EWOULDBLOCKIO;
+            if (atomic)
+                break;
             dentry = real_lookup(nd->dentry, &this, 0);
             err = PTR_ERR(dentry);
```

```
            if (IS_ERR(dentry))
@@ -891,6 +901,8 @@

     if (f & O_DIRECTORY)
         retval |= LOOKUP_DIRECTORY;
+    if (f & O_ATOMICLOOKUP)
+        retval |= LOOKUP_ATOMIC;

     return retval;
 }
--- linux/include/linux/fs.h.orig Tue Apr 17 13:18:51 2001
+++ linux/include/linux/fs.h Tue Apr 17 13:18:52 2001
@@ -1203,6 +1203,7 @@
 #define LOOKUP_POSITIVE      (8)
 #define LOOKUP_PARENT        (16)
 #define LOOKUP_NOALT         (32)
+#define LOOKUP_ATOMIC        (64)
 /*
  * Type of the last component on LOOKUP_PARENT
  */
--- linux/include/asm-i386/fcntl.h.orig   Fri Sep 22 23:21:19 2000
+++ linux/include/asm-i386/fcntl.h    Tue Apr 17 13:18:52 2001
@@ -20,6 +20,7 @@
 #define O_LARGEFILE 0100000
 #define O_DIRECTORY 0200000 /* must be a directory */
 #define O_NOFOLLOW  0400000 /* don't follow links */
+#define O_ATOMICLOOKUP  01000000 /* do atomic file lookup */

 #define F_DUPFD     0   /* dup */
 #define F_GETFD     1   /* get close_on_exec */
--- linux/include/asm-alpha/fcntl.h.orig  Sun Oct  8 18:04:04 2000
+++ linux/include/asm-alpha/fcntl.h   Tue Apr 17 13:18:52 2001
@@ -21,6 +21,8 @@
 #define O_DIRECTORY 0100000 /* must be a directory */
 #define O_NOFOLLOW  0200000 /* don't follow links */
 #define O_LARGEFILE 0400000 /* set by the kernel on every open */
+#define O_ATOMICLOOKUP  01000000 /* do atomic file lookup */
+

 #define F_DUPFD     0   /* dup */
 #define F_GETFD     1   /* get close_on_exec */
--- linux/include/asm-sparc/fcntl.h.orig  Tue Oct 10 19:33:52 2000
+++ linux/include/asm-sparc/fcntl.h   Tue Apr 17 13:18:52 2001
@@ -20,6 +20,7 @@
 #define O_DIRECTORY 0x10000 /* must be a directory */
 #define O_NOFOLLOW  0x20000 /* don't follow links */
 #define O_LARGEFILE 0x40000
+#define O_ATOMICLOOKUP  0x80000 /* do atomic file lookup */

 #define F_DUPFD     0   /* dup */
 #define F_GETFD     1   /* get close_on_exec */
--- linux/include/asm-sparc64/fcntl.h.orig    Tue Oct 10 19:33:52 2000
```

```
+++ linux/include/asm-sparc64/fcntl.h Tue Apr 17 13:18:52 2001
@@ -20,6 +20,7 @@
 #define O_DIRECTORY 0x10000 /* must be a directory */
 #define O_NOFOLLOW  0x20000 /* don't follow links */
 #define O_LARGEFILE 0x40000
+#define O_ATOMICLOOKUP  0x80000 /* do atomic file lookup */

 #define F_DUPFD     0   /* dup */
 #define F_GETFD     1   /* get close_on_exec */
--- linux/include/asm-ia64/fcntl.h.orig   Tue Oct 10 02:54:58 2000
+++ linux/include/asm-ia64/fcntl.h    Tue Apr 17 13:18:52 2001
@@ -28,6 +28,7 @@
 #define O_LARGEFILE 0100000
 #define O_DIRECTORY 0200000 /* must be a directory */
 #define O_NOFOLLOW  0400000 /* don't follow links */
+#define O_ATOMICLOOKUP  01000000 /* do atomic file lookup */

 #define F_DUPFD     0   /* dup */
 #define F_GETFD     1   /* get close_on_exec */
```

I claim: